A.2 Python Built-In Data Types Reference

Adapted from https://docs.python.org/3/library/stdtypes.html.

Boolean type — `bool`

Boolean values are the two constant objects `False` and `True`. They are used to represent truth values.

Numeric types — `int`, `float`

There are two distinct numeric types: integers and floating point numbers. Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point. Comparisons between numbers of mixed type use the same rule.

All numeric types support the following operations (for priorities of the operations, see Operator precedence):

Operation	Description
`x + y`	Returns the sum of `x` and `y`.

```
>>> x = 5
>>> y = 3
>>> x + y
8
```

Operation	Description
`x - y`	Returns the difference of `x` and `y`. `>>> x = 5` `>>> y = 3` `>>> x - y` `2`
`x * y`	Returns the product of `x` and `y`. `>>> x = 5` `>>> y = 3` `>>> x * y` `15`
`x / y`	Returns the quotient of `x` and `y`. `>>> x = 5` `>>> y = 3` `>>> x / y` `1.6666666666666667`
`x // y`	Returns the floored quotient of `x` and `y`. Also referred to as *integer division*. `>>> x = 5` `>>> y = 3` `>>> x // y` `1`
`x % y`	Returns the remainder of `x / y`. `>>> x = 5` `>>> y = 3` `>>> x % y` `2`

Operation	Description
`x ** y`	Returns `x` to raised to the power of `y`. `>>> 3 ** 2` `9`
`-x`	Returns `x` negated. `>>> x = 5` `>>> -x` `-5`
`int(x)`	Returns `x` converted to integer. For floating-point numbers, this truncates towards 0. `>>> x = '3'` `>>> int(x)` `3` `>>> x = -1.5` `>>> int(x)` `-1`
`float(x)`	Returns `x` converted to a floating point number. `>>> x = '3.5'` `>>> float(x)` `3.5` `>>> x = -1` `>>> float(x)` `-1.0`
`math.floor(x)`	Returns the greatest integer `<= x`. `>>> import math` `>>> math.floor(2.45)` `2`

Operation	Description
`math.ceil(x)`	Returns the least integer `>= x`.

```
>>> import math
>>> math.ceil(2.45)
3
```

See also the built-in functions `abs`, `divmod`, `pow`, and `round`.

Sequence types — `str`, `list`, `tuple`

The operations in the following table are supported by most sequence types, both mutable and immutable.

Operation	Description
`x in s`	Returns `True` if an item of `s` is equal to `x`, else `False`.

```
>>> s = ['Mon', 'Wed', 'Fri']
>>> x = 'Tue'
>>> x in s
False
>>> x = 'Wed'
>>> x in s
True
```

`x not in s`	Returns `False` if an item of `s` is equal to `x`, else `True`.

```
>>> s = ['Mon', 'Wed', 'Fri']
>>> x = 'Tue'
>>> x not in s
True
>>> x = 'Wed'
>>> x not in s
False
```

Operation	Description
`s + t`	Returns the concatenation of `s` and `t`. `>>> s = 'Mon'` `>>> x = 'day'` `>>> x + s` `'Monday'`
`s * n` or `n * s`	Returns the equivalent to adding `s` to itself `n` times. `>>> s = 'ha'` `>>> x = 5` `>>> x * s` `'hahahahaha'`
`s[i]`	Returns the `i`th item of `s`, with starting index `0`. `>>> s = 'Hugo'` `>>> s[0]` `'H'` `>>> s[-1]` `'o'`
`s[i:j]`	Returns the slice of `s` from `i` to `j`. `>>> s = 'Hugo'` `>>> s[0:3]` `'Hug'` `>>> s[-2:]` `'go'`

Operation	Description
`s[i:j:k]`	Returns the slice of `s` from `i` to `j` with step `k`. `>>> s = 'Hugo'` `>>> s[0:3:2]` `'Hg'` `>>> s[-1:-4:-2]` `'ou'` `>>> s[::-1]` `'oguH'`
`s.index(x[, i[, j]])`	Returns the `index` of the first occurrence of `x` in `s` (at or after index `i` and before index `j`, if those parameters are given). `>>> s = 'Bipbopboopbap'` `>>> s.index('b')` `3` `>>> s.index('b', 3)` `3` `>>> s.index('b', 4)` `6` `>>> s.index('b', 0, 5)` `3`
`s.count(x)`	Returns the total number of occurrences of `x` in `s`. `>>> s = 'Bipbopboop'` `>>> s.count('o')` `3` `>>> s = ['ho', 'hey', 'ho']` `>>> s.count('ho')` `2` `>>> s.count('h')` `0`

See also the built-in functions `len`, `max`, and `min`.

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length.

Mutable sequence type—`list`

The `list` data type supports all of the immutable sequence operations from the previous section, as well as the following operations.

Operation	Description
`s[i] = x`	Set the item at index `i` of `s` to be `x`. `>>> s = [1, 2, 3, 4, 5]` `>>> s[3] = 100` `>>> s` `[1, 2, 3, 100, 5]`
`list.append(self, x)`	Appends `x` to the end of the sequence `self`. `>>> s = [1, 2, 3, 4, 5]` `>>> list.append(s, 100)  # or, s.append(100)` `>>> s` `[1, 2, 3, 4, 5, 100]`
`list.extend(self, t)` or `self += t`	Extends `self` with the contents of an iterable `t`. `>>> s = [1, 2, 3, 4, 5]` `>>> list.extend(s, [100, 200])  # or, s.extend([100, 200])` `>>> s` `[1, 2, 3, 4, 5, 100, 200]` `>>> s += [300]` `>>> s` `[1, 2, 3, 4, 5, 100, 200, 300]`
`list.insert(self, i, x)`	Inserts `x` into `self` at the index given by `i`. `>>> s = [1, 2, 3, 4, 5]` `>>> list.insert(s, 1, 100)  # or, s.insert(1, 100)` `>>> s` `[1, 100, 2, 3, 4, 5]`

Operation	Description
`list.pop(self[, i])`	Returns the item at `i` and also removes it from `self`. If `i` is omitted, the last item is removed and returned. `>>> s = [1, 2, 3, 4, 5]` `>>> s.pop()  # or, list.pop(s)` `>>> s` `[1, 2, 3, 4]` `>>> s.pop(1)  # or, list.pop(s, 1)` `>>> s` `[1, 3, 4]`
`list.remove(self, x)`	Removes the first occurrence of `x` from `self`. `>>> s = [1, 2, 3, 4, 5]` `>>> list.remove(s, 3)  # or,` `        s.remove(3)` `>>> s` `[1, 2, 4, 5]`
`list.reverse(self)`	Reverses the items of `self` in place (mutates `self`).

Operation	Description
`list.sort(` `  self, *,` `  key=None,` `  reverse=False` `)`	Sorts `self` in place, using only `<` comparisons between items. `key` specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value. `reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed. The `list.sort` method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal—this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade). Example: `>>> s = [1, -1, 0, 2, 3]` `>>> list.sort(s)  # or, s.sort()` `>>> s` `[-1, 0, 1, 2, 3]` `>>> list.sort(s, reverse=True)  # or,` `  s.sort(reverse=True)` `>>> s` `[3, 2, 1, 0, -1]`

Text sequence type—`str`

Textual data in Python is handled with `str` objects, or **strings**. Strings are immutable sequences.

Triple quoted strings may span multiple lines—all associated whitespace will be included in the string literal.

Strings may also be created from other objects using the `str` constructor.

Since there is no separate "character" type, indexing a string produces strings of length 1. That is, for a non-empty string `s`, `s[0] == s[0:1]`. Strings implement all of the common sequence operations, along with the additional methods described below.

Operation	Description

Operation	Description
`str.capitalize(self)`	Return a copy of the string with its first character capitalized and the rest lowercased. Example: `>>> s = ABC'` `>>> str.capitalize(s)  # or, s.capitalize()` `'Abc'` `>>> s = 'aBC'` `>>> str.capitalize(s)` `'Abc'`
`str.count( self, sub [, start [, end]] )`	Return the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Example: `>>> s = 'Beepbopboopbop'` `>>> str.count(s, 'o')  # or, s.count('o')` `4` `>> str.count(s, 'bo')` `3`
`str.endswith( self, suffix [, start [, end]] )`	Return `True` if the string ends with the specified suffix, otherwise return `False` `suffix` can also be a tuple of suffixes to look for. With optional `start`, test beginning at that position. With optional `end`, stop comparing at that position. Example: `>>> s = 'www.google.com'` `>>> str.endswith(s, '.com')  # or, s.endswith('.com')` `True` `>>> s.endswith('.com', 12)` `False`

Operation	Description
`str.find(self, sub[, start[, end]])`	Return the lowest index in the string where substring `sub` is found within the slice `s[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Example: `>>> s = 'www.google.com'` `>>> str.find(s, 'o')  # or, s.find('o')` `5` `>>> s.find('.com')` `10` `>>> s.find(bop')` `-1` Return -1 if `sub` is not found. Note: The `find()` method should be used only if you need to know the position of `sub`. To check if `sub` is a substring or not, use the `in` operator: `>>> 'Py' in 'Python'` `True`
`str.index(` `  self, sub` `  [, start` `  [, end]]` `)`	Like `find()`, but raise `ValueError` when the substring is not found. Example: `>>> s = 'www.google.com'` `>>> str.index(x, 'o')  # or, s.index('o')` `5` `>>> s.index('.com')` `10` `>>> s.index('bop')` `ValueError: substring not found`

Operation	Description
`str.isalnum(self)`	Return `True` if all characters in the string are alphanumeric and there is at least one character, `False` otherwise. A character `c` is alphanumeric if one of the following returns `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`. Example: `>>> s = 'www.google.com'` `>>> str.isalnum(s)  # or, s.isalnum()` `False` `>>> s = 'passw0rd'` `>>> s.isalnum()` `True`
`str.isalpha(self)`	Return `True` if all characters in the string are alphabetic and there is at least one character, `False` otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that this is different from the “Alphabetic” property defined in the Unicode Standard. Example: `>>> s = ''` `>>> str.isalpha(s)  # or, s.isalpha()` `False` `>>> s = 'passw0rd'` `>>> s.isalpha()` `False` `>>> s = 'word'` `>>> s.isalpha()` `True`

Operation	Description
`str.isdigit(self)`	Return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Example: `>>> s = '10'` `>>> str.isdigit(s)  # or, s.isdigit()` `True` `>>> s = '-10'` `>>> s.isdigit()` `False` `>>> s = '10 kittens'` `>>> s.isdigit()` `False`
`str.islower(self)`	Return `True` if all cased characters in the string are lowercase and there is at least one cased character, `False` otherwise. Example: `>>> s = 'www.google.com'` `>>> str.islower(s)  # or, s.islower()` `True` `>>> s = 'Capitalized'` `>>> s.islower()` `False`
`str.isnumeric(self)`	Return `True` if all characters in the string are numeric characters, and there is at least one character, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property. Example: `>>> s = 'abc123'` `>>> str.isnumeric(s)  # or,` `s.isnumeric()` `False` `>>> s = '1647123123'` `>>> s.isnumeric()` `True`

Operation	Description
`str.isupper(self)`	Return `True` if all cased characters in the string are uppercase and there is at least one cased character, `False` otherwise. Example: ``` >>> s = 'www.google.com' >>> str.isupper(s) # or, s.isupper() False >>> s = 'Capitalized' >>> s.isupper() False >>> s = 'SHOUTING' >>> s.isupper() True ```
`str.join(self, iterable)`	Return a string which is the concatenation of the strings in `iterable`. A `TypeError` will be raised if there are any non-string values in `iterable`, including bytes objects. The separator between elements is the string providing this method. Example: ``` >>> lst = ['John', 'David', 'Jen'] >>> separator = ', and ' >>> str.join(separator, lst) # or, separator.join(lst) 'John, and David, and Jen' ```
`str.lower(self)`	Return a copy of the string with all the cased characters converted to lowercase. Example: ``` >>> s = 'lower' >>> str.lower(s) # or, s.lower() 'www.google.com' >>> s = 'Capitalized' >>> s.lower() 'capitalized' >>> s = 'SHOUTING' >>> s.lower() 'shouting' ```

Operation	Description
`str.replace(` `  self, old, new` `  [, count]` `)`	Return a copy of the string with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced. Example: `>>> s = 'www.google.com'` `>>> str.replace(s, 'www.',` `'https://')  # or,` `s.replace('www.',` `'https://')` `'https://google.com'` `>>> s = 'Far Farquad on a Far Quad'` `>>> s.replace('Far', 'Close')` `'Close Closequad on a Close Quad'`

Operation	Description
`str.split(` `  self,` `  sep=None,` `  maxsplit=-1` `)`	Return a list of the words in the string, using `sep` as the delimiter string. If `maxsplit` is given, at most `maxsplit` splits are done (thus, the list will have at most `maxsplit+1` elements). If `maxsplit` is not specified or -1, then there is no limit on the number of splits (all possible splits are made). If `sep` is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The `sep` argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns ['']. Example: `>>> str.split('1,2,3', ',')  # or, '1,2,3'.split(',')` `['1', '2', '3']` `>>> '1,2,3'.split(',', maxsplit=1)` `['1', '2,3']` `>>> '1,2,,3,'.split(',')` `['1', '2', '', '3', '']` If `sep` is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.
`str.startswith(` `  self, prefix,` `  [, start` `  [, end]]` `)`	Return `True` if string starts with the `prefix`, otherwise return `False`. With optional `start`, test string begins at that position. With optional `end`, stop comparing string at that position.

Operation	Description
`str.strip(self, [chars])`	Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the chars argument defaults to removing whitespace. The `chars` argument is not a prefix or suffix; rather, all combinations of its values are stripped. Example: `>>> str.strip('  spacious  ') # or, '  spacious  '.strip()` `'spacious'` `>>> 'www.example.com'.strip('cmowz.')` `'example'` The outermost leading and trailing chars argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in chars. A similar action takes place on the trailing end. Example: `>>> comment_string = '#.....Section 3.2.1 Issue #32.....'` `>>> comment_string.strip('.#! ')` `'Section 3.2.1 Issue #32'`

Set type—`set`

A set object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

Like other collections, sets support x in set, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

The set type is mutable—the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary

key or as an element of another set.

Non-empty sets can be created by placing a comma-separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the `set` constructor.

Operation	Description
`len(self)`	Return the size (number of elements) of `self`.
`x in self`	Return whether `x` is in `self`.
`x not in self`	Return whether `x` is *not* in `self`.
`set.isdisjoint(self, other)`	Return whether the set `self` has no elements in common with `other`. Sets are disjoint if and only if their intersection is the empty set.
`set.issubset(self, other)`	Return whether every element in the set `self` is in `other`. Can also use `self <= other`.
`self < other`	Return whether the set `self` is a proper subset of `other`, that is, `self <= other and self != other`.
`set.issuperset(self, other)`	Return whether every element in `other` is in the set `self`. Can also use `self >= other`.
`self > other`	Return whether the set `self` is a proper superset of `other`, that is, `self >= other and self != other`.
`set.union(self, *others)`	Return a new set with elements from the set and all others.
`set.intersection(self, *others)`	Return a new set with elements common to the set and all others.
`set.difference(self, *others)`	Return a new set with elements in the set that are not in the others.
`set.symmetric_difference(self, other)`	Return a new set with elements in either the set or `other` but not both.
`set.update(self, *others)`	Update the set, adding elements from all others.

Operation	Description
`set.intersection_update(self, *others)`	Update the set, keeping only elements found in it and all others.
`set.difference_update(self, *others)`	Update the set, removing elements found in others.
`set.symmetric_difference_update(self, other)`	Update the set, keeping only elements found in either set, but not in both.
`set.add(self, elem)`	Add element `elem` to the `set`.
`set.remove(self, elem)`	Remove element `elem` from the set. Raises `KeyError` if `elem` is not contained in the set.
`set.discard(self, elem)`	Remove element `elem` from the set if it is present.
`set.pop(self)`	Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

`set` supports set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Mapping type–`dict`

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary.

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`, or by the `dict` constructor.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

Operation	Description
`list(d)`	Return a list of all the keys used in the dictionary `d`.
`len(d)`	Return the number of items in the dictionary `d`.
`d[key]`	Return the item of `d` with key `key`. Raises a `KeyError` if `key` is not in the map.
`d[key] = value`	Set `d[key]` to `value`.

Operation	Description
`key in d`	Return `True` if `d` has a key `key`, else `False`.
`key not in d`	Equivalent to `not key in d`.
`dict.get(self, key [, default])`	Return the value for `key` if `key` is in the dictionary, else `default`. If `default` is not given, it defaults to `None`, so that this method never raises a `KeyError`.
`dict.items(self)`	Return a new view of the dictionary's items ((key, value) pairs).
`dict.pop(self, key [, default])`	If `key` is in the dictionary, remove it and return its value, else return `default` If `default` is not given and `key` is not in the dictionary, a `KeyError` is raised.
`dict.popitem(self)`	Remove and return a (key, value) pair from the dictionary. Pairs are returned in last-in-first-out (LIFO) order. `popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.
`dict.setdefault(self, key [, default])`	If `key` is in the dictionary, return its value. If not, insert `key` with a value of `default` and return `default`. `default` defaults to `None`.

Dictionaries compare equal if and only if they have the same (key, value) pairs (regardless of ordering). Order comparisons (<, <=, >= >) raise `TypeError`.

Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Numeric sequence data type—`range`

The `range` type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

Constructor: `range(stop)` or `range(start, stop[, step])`.

The arguments to the range constructor must be integers. If the `step` argument is omitted, it defaults to 1. If the `start` argument is omitted, it defaults to 0. If `step` is zero, `ValueError` is raised.

For a positive step, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative step, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

Range examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

The "Null" type—`None`

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name).